Capstone Project 1

Predicting demand for cycle hire scheme

Author: Vanita Kalaichelvan

November 29, 2019

Contents

1 Introduction **3**

1.1 Problem Statement 3

1.2 Description of dataset 4

1.2.1 Data wrangling 4

1.2.2 Feature Engineering 5

2 Exploratory Analysis **6**

2.1 Weather 6

2.2 Type of Day 7

2.3 Hour of Day 7

2.4 Ride count vs duration 8

3 Appendix **9**

3.1 Part 1: Data Wrangling 9

3.2 Part 2: Data Storytelling 29

3.3 Part 3 & 4: Feature Engineering & Statistical Analysis 38

Chapter 1

Introduction

1.1 Problem Statement

The pubic bike hire scheme is a great way of maximising the efficiency of public transportation systems while reducing pollution. In London, the Sanatander cycle hire scheme was started in 2010 (originally sponsored by Barclays) and it continues to be an envionmentaly friendly and healthy way to explore the city or just commute. Another useful aspect of the scheme is that bikes can be hired at any time of the day and there are 839 stations densely populated around central London.

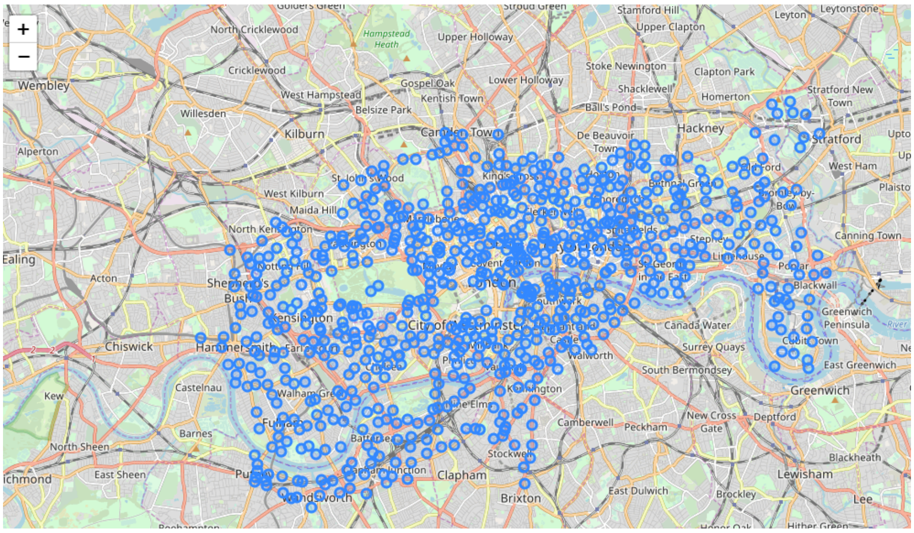


Figure 1.1: Location of cycle stations

In recent years, more players are entering the cycle hire scheme such as Mobike and Ofo that offer lower pricing for short trips under 30 minutes with a more seamless experience that doesn't require docking. While we cannot change the way the bikes are tracked, we can improve overall output of the system by matching availability with demand and identifying under utilized or over utilized spaces. In a recent survey done about the public cycle hire scheme, the key consumer concerns were bike availability and space availability at docking stations (see quarterly reports released by the scheme)

In this project, we will use historical data provided by the trip data to build a model that will predict the upcoming day's ride count and average duration based on historical observations. This is known as time series forecasting.

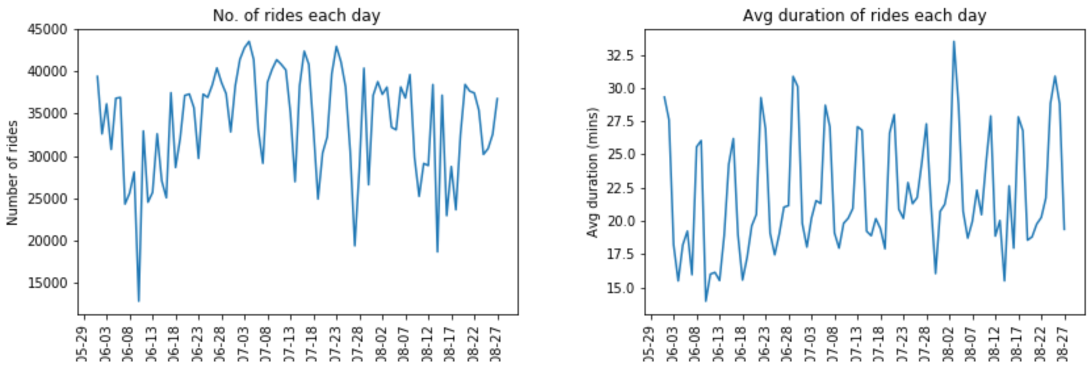


Figure 1.2: Number of rides and average duration per day

1.2 Description of dataset

The dataset is provided by TFL (Transport for London) which has a record of every trip made. An example of the dataset can be seen in Figure 1.3. You can also refer to 3.1, which shows how the data is downloaded from the AWS file storage system, S3. The easiest way is to use boto 3 to access and read each file into a string, concatenate the strings into a large string object and then, write the object to a pandas dataframe. This helps to save memory space and improves time performance.

	Rental Id	Duration	Bike Id	End Date	EndStation Id	EndStation Name	Start Date	StartStation Id	StartStation Name
0	50754225	240	11834	2016-01-10 00:04:00	383.0	Frith Street, Soho	2016-01-10 00:00:00	18	Drury Lane, Covent Garden
1	50754226	300	9648	2016-01-10 00:05:00	719.0	Victoria Park Road, Hackney Central	2016-01-10 00:00:00	479	Pott Street, Bethnal Green
2	50754227	1200	10689	2016-01-10 00:20:00	272.0	Baylis Road, Waterloo	2016-01-10 00:00:00	425	Harrington Square 2, Camden Town

Figure 1.3: Sample data of each trip

As seen within the dataset, each row refers to one trip with information such as date, time, starting point, ending point and duration of ride. The dataset runs for 87 days from 01/06/2019 to 27/08/2019.

1.2.1 Data wrangling

You can refer to 3.1 for the detailed description of the data wrangling methods. To summarize, the following steps were taken:

1. Check all columns are of the right type
2. Check for null values
3. Check values are consistent e.g. station IDs matched with station names, duration matched with difference in time
4. Check for any outliers

From step 1 and 2, we see that the date time values are recognized correctly and that there are no null values. However, we notice that there are some large outliers in the dataset and some of the station names and IDs are not matched due to discrepancies in spacing and punctuation. We also notice that some of the stations do not exist on the station dataset provided by the company, hence we remove these assuming errors in observations which removes $<2\%$ of our dataset. As for naming differences, we simply fix them for consistency and decide to keep the outliers as it most likely reflects the behaviour of not returning the cycles properly than an actual recording error.

1.2.2 Feature Engineering

This section is covered partly in 3.1 and 3.3. For the dataset to be provide more useful information for better prediction, we can add more features. In this case, we add the hour of day (0-24), type of day (weekday/weekend) and weather conditions as additional features. While the first feature is easily extracted from the dataset, the other 2 require a bit more work. The day of week (Mon-Sun) can be found using datetime functions but it doesn't tell us anything about public holidays. Here, we use the `holidays` package to also identify UK holidays.

As for weather information, we can pull historical data and forecasts from a weather API for the given dates. We get temperature, wind speed and a categorical variable for overall condition for every half hour or hour. Once again, we need to go through the data wrangling steps before we can merge this data with our feature dataset. The only issues are the outliers in wind speed that seem to be most likely an error in recording. Instead of just removing the data, we manage these by interpolating between available clean data. After cleaning the data, the final problem to tackle is the categorical weather data with 26 different categories. Most machine learning models will not perform well on categorical data, especially one with so many variants. This is dealt with by shrinking the categories to 4 main ones, namely good weather, ok weather, bad weather and very bad weather. We can then encode these categories into binary variables and merge this data with our main dataset.

Finally, we reduce our dataset to give daily aggregations (see 3.3. This is so that the model can predict next day demand based on previous days' data. We also add 3 more features: the number of bikes taken from each station the day before, the number of bikes docked at the station the day before and the 7 day rolling average of the duration of rides starting at the station. The possible Y variables to predict are number of rides starting at a station on a given day, the number of rides ending at a station on a given day and the average duration of rides starting at the station on a given day. The final features can be seen in 1.1

		day_no	is_weekday	start_count_day_bf	end_count_day_bf	7d_rolling_dur	temperature	wind_speed	good_weather	ok_weather	bad_weather	very_bad_weather
StartStation Id	Date											
	2019-06-09	6	0	19.0	9.0	12.701900	15.111111	11.777778	0.703704	0.296296	0.000000	0.0
	2019-06-10	0	1	27.0	17.0	12.746154	13.375000	17.750000	0.125000	0.375000	0.500000	0.0
	2019-06-11	1	1	8.0	5.0	13.381408	14.250000	11.300000	0.850000	0.150000	0.000000	0.0
1	2019-06-12	2	1	20.0	14.0	13.483942	12.903226	7.806452	0.741935	0.161290	0.096774	0.0
	2019-06-13	3	1	31.0	19.0	13.189248	14.095238	28.523810	0.619048	0.380952	0.000000	0.0
	2019-06-14	4	1	21.0	14.0	13.170451	16.550000	23.550000	1.000000	0.000000	0.000000	0.0
	2019-06-15	5	1	20.0	16.0	14.681730	17.681818	23.909091	1.000000	0.000000	0.000000	0.0

Table 1.1: Features for model

Chapter 2

Exploratory Analysis

2.1 Weather

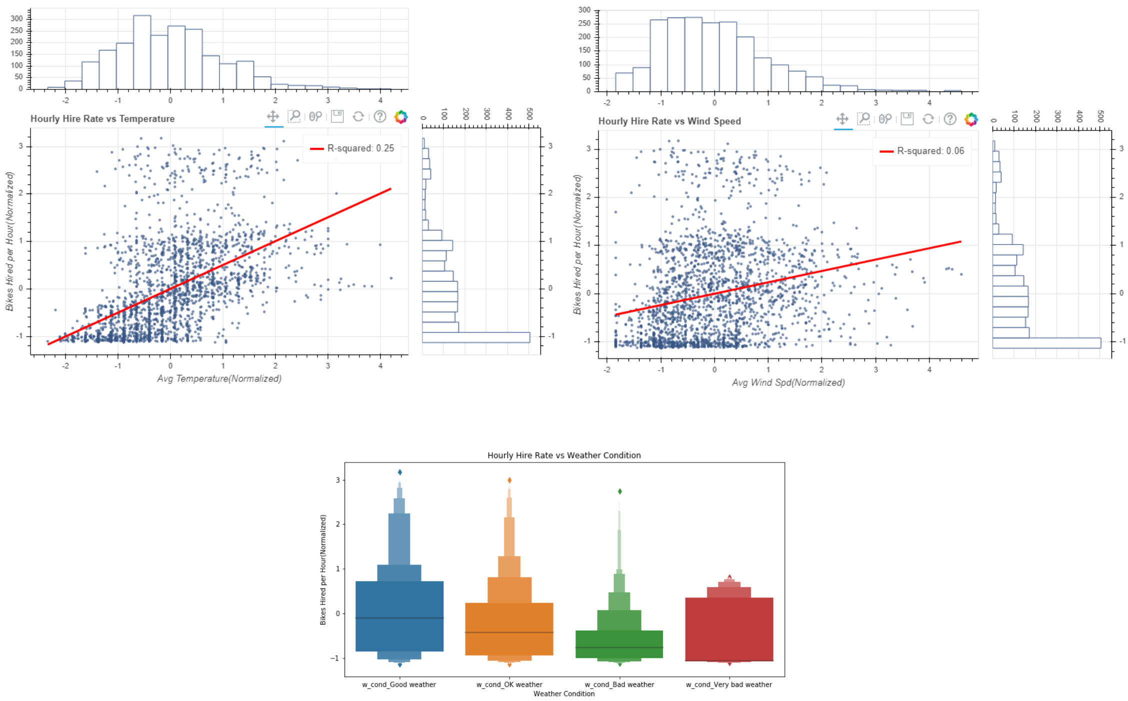


Figure 2.1: Relationship between weather conditions and number of rides

The relationship between temperature and ride count is significant and the number of rides increase with temperature. The relationship between wind speed and ride count is less significant but there still seems to be a weak positive correlation. As for the categorical weather conditions, it is clear that the ride count decreases significantly with worsening weather. There are some outliers that skew the data significantly but the median values clearly show a negative relationship.

2.2 Type of Day

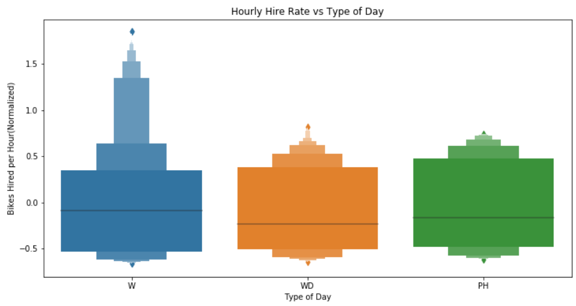


Figure 2.2: Relationship between type of day and number of rides

Weekdays seem to have higher frequency of bike hires and a greater spread of hourly hires.

2.3 Hour of Day

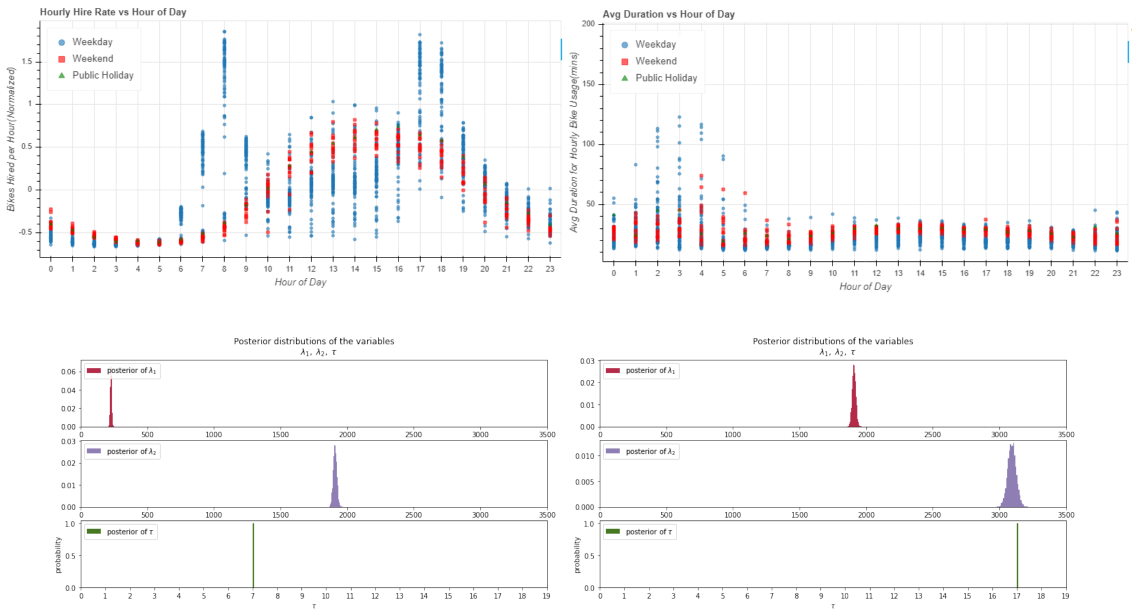


Figure 2.3: Relationship between hour of day and ride count and ride duration

The hour of day clearly has an influence on the number of bikes hires. We can see that there are higher peaks on weekdays around peak hours while weekends have a more normal distribution with a mean around 2pm. As for the duration of rides, they seem to be higher during night time when the frequency is at its lowest. We can also use bayesian inference to understand when during the day the hiring of bikes change. The analysis tells us that there is change in frequency of bike hires around 7am and 5pm which coincides with peak hours in London (when people leave for work and come back from work).

2.4 Ride count vs duration

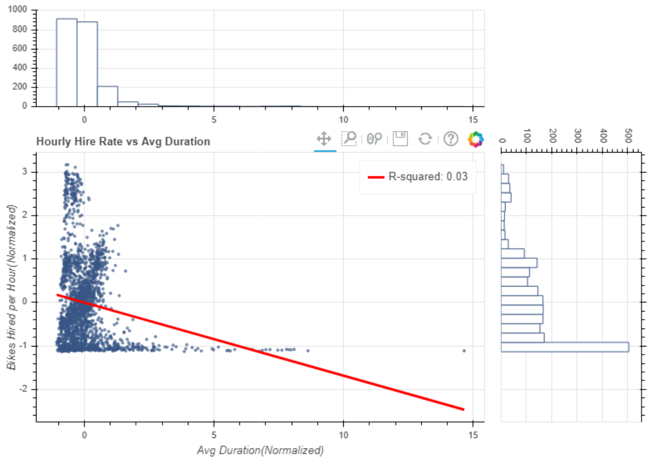


Figure 2.4: Relationship between ride count and duration of rides

We can also look to see if the frequency of bike hires is inversely proportional to the duration as observed in 2.3. There seems to be a very weak negative relationship which is not enough to support our initial observations.

Chapter 3

Appendix

3.1 Part 1: Data Wrangling

Capston Project Part 1: Data Wrangling

Vanita Kalaichelvan

The first part of the capstone project involves cleaning the data and adding appropriate features that will be useful in creating a model for the cycle hire scheme. Firstly, we need to retrieve the data from the AWS S3 file storage system. Here, we use the package, `boto3` to access the files. The function `find_bucket_obj()` retrieves the name of all the files stored in the S3 bucket and the funtion `s3_files_to_df()` reads the files into a string object and then, parses it into a dataframe.

[2]:
```
import logging
import boto3
import re
import pandas as pd
import numpy as np
from
time import datetime
from io import StringIO
from botocore.exceptions import ClientError
from aws_keys import ACCESS_KEY, SECRET_KEY


def find_bucket_obj(bucket_name, ACCESS_KEY, SECRET_KEY):
    """find all objects in AWS S3 bucket"""

    s3 = boto3.client('s3', aws_access_key_id=ACCESS_KEY,
     aws_secret_access_key=SECRET_KEY)

    try:
        response = s3.list_objects_v2(Bucket=bucket_name)
    except ClientError as e:
    # AllAccessDisabled error == bucket not found
        logging.error(e)
        return None

    return response


def s3_files_to_df(bucket_name, key_names,ACCESS_KEY, SECRET_KEY):
    """appends S3 files into a dataframe"""

    s3 = boto3.client('s3', aws_access_key_id=ACCESS_KEY,
     aws_secret_access_key=SECRET_KEY)
```

```
    #quicker way to append files than appending straight into df
    concat = StringIO()
    headers = StringIO()
    for i, key in enumerate(key_names):
        file = s3.get_object(Bucket=bucket_name, Key=key)
        string_obj = file['Body'].read().decode('utf-8')
        concat.write(string_obj[112:])

    headers = string_obj[:112].split('\r\n')[0].split(',') #set column names
    data_type = {0:np.int64, 1:np.int64, 2:np.int64, 4:np.int64, 7:np.int64}
    dateparser = lambda x: pd.datetime.strptime(x, "%d/%m/%Y %H:%M")

    concat.seek(0) #bring file pointer back to 0
    df = pd.read_csv(concat, dtype=data_type, parse_dates=[3, 6],␣
↪date_parser=dateparser, header=None,
                names=headers)

    return df


bucket_name = 'cycling.data.tfl.gov.uk'

response = find_bucket_obj(bucket_name, ACCESS_KEY, SECRET_KEY)

#find files in bucket that are of type csv and under usage-stats folder
key_names = (bucket_dict['Key'] for bucket_dict in response['Contents']
                        if re.search("\Ausage-stats.*19.csv",␣
↪bucket_dict['Key']))

cycle_files_df = s3_files_to_df(bucket_name, key_names, ACCESS_KEY, SECRET_KEY)
```

Now that we have downloaded the files, we first want to check if the dataset is clean and if not, use data wrangling to clean it such that we can use it in our model. The first things to check for are:

- Null values
- Station names are matched with station IDs
- Station names are all valid (can be found in the dock locations file)
- Durations are all matched
- Time Outliers

Depending on the outcome, we can either choose to remove certain data points completely or fill missing/incorrect values based on what information we have.

```
[3]: cycle_files_df = cycle_files_df[cycle_files_df['Start Date'] >= datetime.
      ↪strptime('01/06/19', '%d/%m/%y')]
     cycle_df = cycle_files_df.sort_values(by=['Start Date'])
     cycle_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3032277 entries, 4066853 to 7005695
Data columns (total 9 columns):
Rental Id             int64
Duration              int64
Bike Id               int64
End Date              datetime64[ns]
EndStation Id         int64
EndStation Name       object
Start Date            datetime64[ns]
StartStation Id       int64
StartStation Name     object
dtypes: datetime64[ns](2), int64(5), object(2)
memory usage: 231.3+ MB
```

```
[4]: diff = cycle_df['Start Date'].max() - cycle_df['Start Date'].min()
     print('The dataset runs from ' + cycle_df['Start Date'].min().strftime('%d/%m/
      ↪%Y') + ' to '
             + cycle_df['Start Date'].max().strftime('%d/%m/%Y') + ' which is ' +␣
      ↪str(diff.days) + ' days.')
```

```
The dataset runs from 01/06/2019 to 27/08/2019 which is 87 days.
```

The dataframe infromation tells us that all the columns are of the desired data type. This is because we have correctly parsed the dates within the `read_csv()` function. Moreover, we can see that there are no null values which is great! Let's now check if all the station names are valid and remove datapoints with invalid station name Then, we will check if the station names and IDs are matched

```
[5]: # import file containing locaitons of all docks
     location_df = pd.read_csv("cycle_dock_locations.csv")

     # drop invalid station names
     cycle_df = cycle_df.drop(cycle_df[~cycle_df['StartStation Id'].
      ↪isin(location_df['id'])].index)
     cycle_df = cycle_df.drop(cycle_df[~cycle_df['EndStation Id'].
      ↪isin(location_df['id'])].index)

     del_no = -len(cycle_df) + 3032277
     print('We have removed {0:,} entries which is {1:.1f}% of entries'.
      ↪format(del_no, (del_no/3379793)*100))
```

```
We have removed 51,960 entries which is 1.5% of entries
```

```
[6]: # check station names and IDs are matched
location_df = location_df.set_index('id')
df = cycle_df[['StartStation Id', 'StartStation Name']].
 ↪merge(location_df['name'],
                                how='left', left_on='StartStation Id',␣
 ↪right_index=True)

map_names = df[df['StartStation Name'] != df['name']].drop_duplicates()
map_names
```

```
[6]:         StartStation Id                      StartStation Name  \
4014729              553             Regent's Row , Haggerston
4084466              832               Ferndale Road, Brixton.
4224504              463               Thurtle road, Haggerston
4333723              725  Thessaly Road North, Wandsworth Road

                              name
4014729  Regent's Row , Haggerston
4084466     Ferndale Road, Brixton
4224504    Thurtle Road, Haggerston
4333723  Walworth Square, Walworth
```

We see 4 names that have issues with matching the actual name. The first 3 are still correct information but get flagged due to character differences. The last one is a completely different entry which we will drop given we have no additional information on how to reconcile the difference.

```
[7]: # map station names from location file to cycle hire data file and replace with␣
 ↪correct name or remove if name non-existent
map_names = map_names[['StartStation Name', 'name']].set_index('StartStation␣
 ↪Name')['name'].to_dict()
map_names['Thessaly Road North, Wandsworth Road'] = np.nan

cycle_df['StartStation Name'] = cycle_files_df['StartStation Name'].
 ↪replace(map_names)
cycle_df = cycle_df.dropna()

df = cycle_df[['StartStation Id', 'StartStation Name']].
 ↪merge(location_df['name'],
                                how='left', left_on='StartStation Id',␣
 ↪right_index=True)

if(df[df['StartStation Name'] != df['name']].empty):
    print('Success. All names match to ID')
```

```
Success. All names match to ID
```

Now that we have sorted out the station names and IDs, we can check if the data has any outliers. This would be signified either by a ride with a very high duration or with a ride with no duration.

We also need to make sure that the duration has been computed correctly.

[8]:
```
print('{} rides have no duration'.format(len(cycle_df[cycle_df['Duration'] ==␣
↪0])))
```

```
0 rides have no duration
```

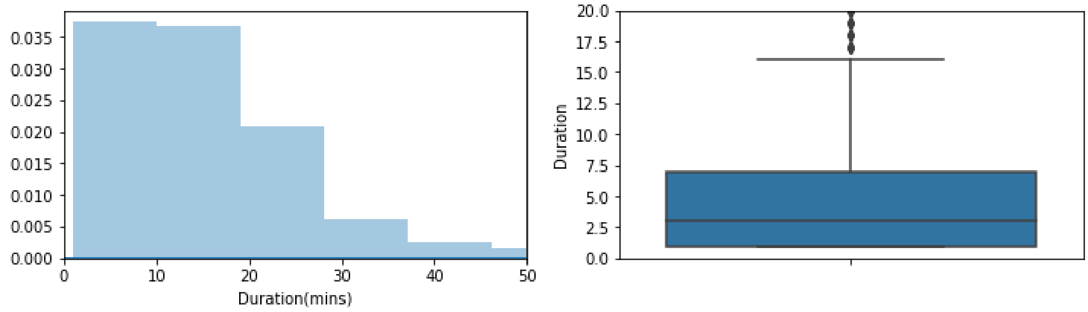
[9]:
```
#check if timedelta between start and end matches duration
check_duration = round((cycle_df['End Date'] - cycle_df['Start Date']).dt.
↪total_seconds())
check_duration = check_duration.apply(lambda x: int(x))
print('Durations are all matched') if check_duration.
↪equals(cycle_df['Duration']) else print("Durations don't match")
```

```
Durations are all matched
```

[10]:
```
# print some statistics about duration
cycle_df['Duration(mins)'] = cycle_df['Duration'].apply(lambda x: x/60)
print('Avg time of rides was {:.0f} mins'.format(cycle_df['Duration(mins)'].
↪mean()))
print('Max time of a ride was {:.0f} mins'.format(cycle_df['Duration(mins)'].
↪max()))
print('Min time of a ride was {:.0f} min'.format(cycle_df['Duration(mins)'].
↪min()))
print('Std deviation between rides was {:.0f} mins'.
↪format(cycle_df['Duration(mins)'].std()))
print('{} rides over the period lasted more than one day'.
↪format(cycle_df[cycle_df['Duration(mins)'] > (24*60)]['Rental Id'].count()))
```

```
Avg time of rides was 22 mins
Max time of a ride was 9024 mins
Min time of a ride was 1 min
Std deviation between rides was 68 mins
807 rides over the period lasted more than one day
```

[13]:
```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(12, 3))
plt.subplot(121)
plt.xlim(right=50)
sns.distplot(cycle_df['Duration(mins)'], bins=1000)

plt.subplot(122)
plt.ylim(top=20)
sns.boxplot(y='Duration', data=cycle_df.groupby('Duration(mins)').count())

plt.show()
```



There are quite a few rides that lasted over a day with the longest one being almost 6 days. This is most likely a result of forgetting to return the bikes or forgetting to dock them regularly throughout the trip. We will leave these outliers in the dataset as this affects the availability of bike at dock stations. It's also obvious from the boxplots that the most ride durations are concentrated below 10 minutes.

Now that we have filtered out unwanted rows, let's clean up the dataset by removing unwanted rows and reorganizing the columns.

[14]:
```
new_cols = ['Start Date', 'StartStation Name', 'End Date', 'EndStation Name',␣
 ↪'Duration(mins)',
              'Bike Id', 'StartStation Id', 'EndStation Id']
cycle_df_clean = cycle_df.drop(columns=['Rental Id', 'Duration'])
cycle_df_clean = cycle_df_clean[new_cols].reset_index(drop=True)
cycle_df_clean.head()
```

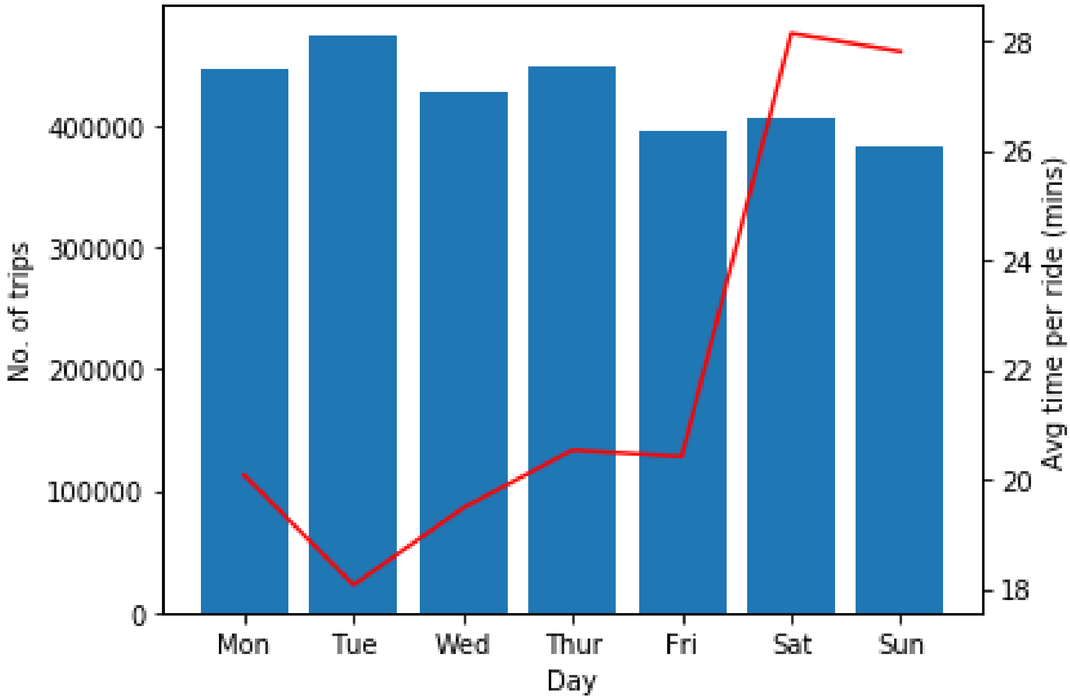
[14]:
```
   Start Date                   StartStation Name            End Date  \
0 2019-06-01  Westminster University, Marylebone 2019-06-01 00:07:00
1 2019-06-01          Upcerne Road, West Chelsea 2019-06-01 00:01:00
2 2019-06-01          Mile End Stadium, Mile End 2019-06-01 00:19:00
3 2019-06-01      Bethnal Green Road, Shoreditch 2019-06-01 00:10:00
4 2019-06-01          Mile End Stadium, Mile End 2019-06-01 00:19:00

                        EndStation Name  Duration(mins)  Bike Id  \
0  St. John's Wood Church, The Regent's Park             7.0    13485
1                Upcerne Road, West Chelsea             1.0    14376
2     Mile End Park Leisure Centre, Mile End            19.0    10693
3                Curlew Street, Shad Thames            10.0     7390
4     Mile End Park Leisure Centre, Mile End            19.0    11332

   StartStation Id  EndStation Id
0              257            247
1              745            745
2              712            763
3              132            298
```

```
4                712              763
```

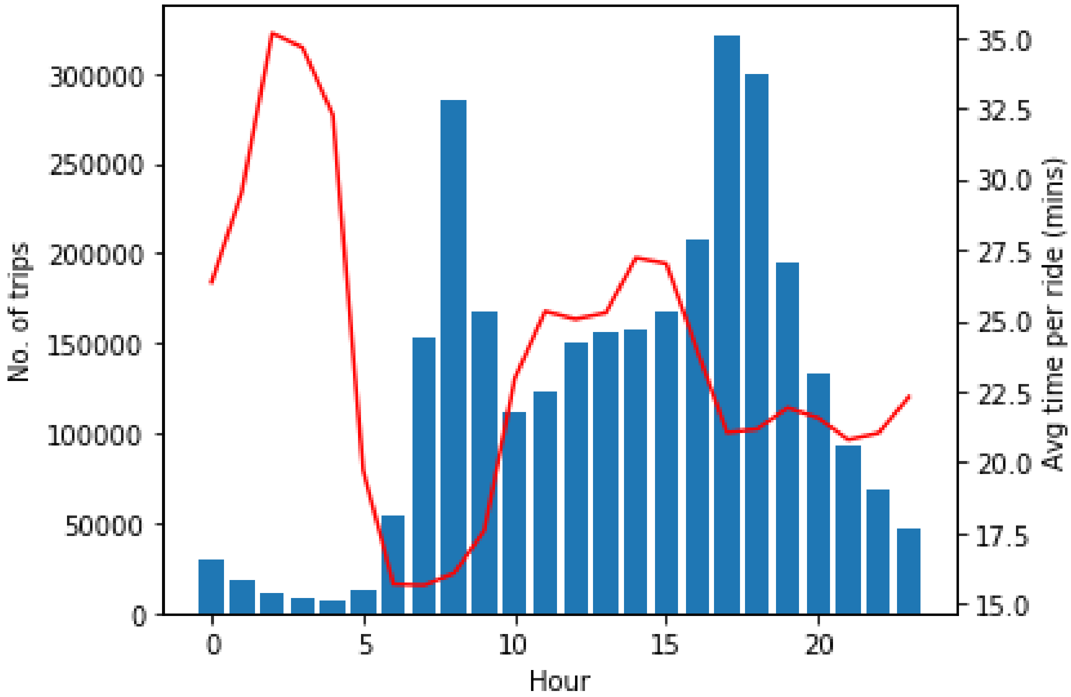
The second stage of data wrangling is to improve the usefulness of the data. We can add more features to our dataset that might be helpful in modelling demand and availability. The obvious factors in hiring a bike are:

- Type of day
- Weather

[15]:
```
def plot_trip_data(x, y1, y2, xlabel, ylabel='No. of trips'):
    """plots count and mean data for trips"""

    fig, ax1 = plt.subplots()

    ax1.set_xlabel(xlabel)
    ax1.set_ylabel(ylabel)
    ax1.bar(x, y1)

    ax2 = ax1.twinx()

    ax2.set_xlabel(xlabel)
    ax2.set_ylabel('Avg time per ride (mins)')
    ax2.plot(x, y2,
             color = 'red')

    fig.tight_layout()
    plt.show()


#plotting number of rides and average time of rides on each weekday
cycle_df_clean['Day'] = cycle_df_clean['Start Date'].dt.weekday
group_by_day = cycle_df_clean.groupby('Day')
day_index = ['Mon','Tue','Wed', 'Thur', 'Fri', 'Sat', 'Sun']
plot_trip_data(day_index, group_by_day.count()['Duration(mins)'],
               group_by_day.mean()['Duration(mins)'], 'Day')
```



[16]:
```
#plotting number of rides and average time of rides for each hour of day
cycle_df_clean['hour'] = cycle_df_clean['Start Date'].dt.hour
groupby_time = cycle_df_clean.groupby('hour')
time_index = groupby_time.sum().index

plot_trip_data(time_index, groupby_time.count()['Duration(mins)'],
               groupby_time.mean()['Duration(mins)'], 'Hour')
```



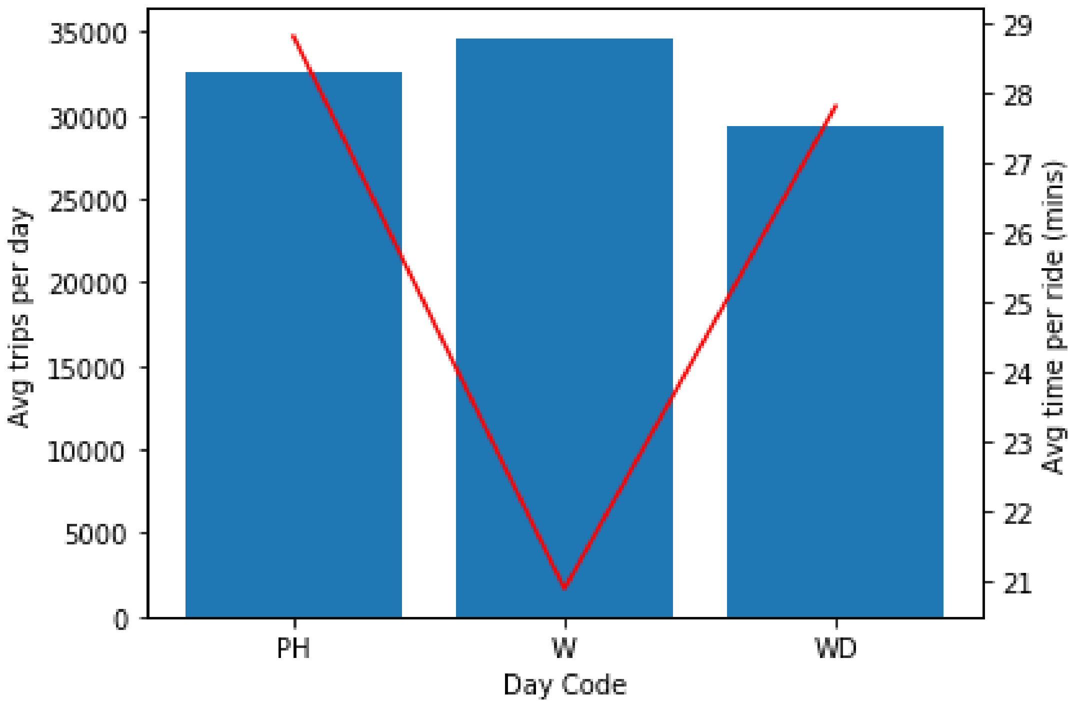
It is clear that there is high demand for bikes druing the weekend as opposed to a workday. We can add an additional feature that identifies if the day is a holiday(weekend/public holiday) or a workday. We use the package `holidays` to get the holidays within the time period.

[37]:
```
import holidays

# find holidays in England that match data time window
ph = list(filter(lambda x: cycle_df_clean.iloc[0]['Start Date'] <= x <=␣
 ↪cycle_df_clean.iloc[-1]['Start Date'],
                 holidays.England(years=2019).keys()))
```

[38]:
```
def set_day_code(row, public_hols):

    if row['Start Date'].date() in public_hols:
        return 'PH'
    elif row['Day'] in [6, 7]:
        return 'WD'
    else:
        return 'W'


# set codes for days based on type of day (weekday=W, weekend=WD, public␣
 ↪holiday=PH)
```

```
cycle_df_clean['day_code'] = cycle_df_clean.apply(lambda x: set_day_code(x, ph),␣
↪axis=1)
```

[39]:
```
# plotting avg trip data based on type of day
cycle_df_clean['Date'] = cycle_df_clean['Start Date'].dt.date
cycle_df_clean = cycle_df_clean.set_index(['Date', 'Start Date'])
plot_trip_data(cycle_df_clean.groupby('day_code').count().index,
               cycle_df_clean.groupby(['day_code', 'Date']).count().
↪groupby('day_code').mean()['Duration(mins)'],
               cycle_df_clean.groupby(['day_code']).mean()['Duration(mins)'],␣
↪'Day Code',
               ylabel='Avg trips per day')
```



Another feature that affects the demand of cycle hires is weather. We can get weather data from a weather API which gives hourly historical data on temperature, wind speed and weather condition from a weather station located in London Southend Airport. We will use the `requests` package to pull data from the API and extract the useful information into a dataframe.

[40]:
```
import requests


def import_weather_from_api(date):
```

```python
    api_url = "https://api.weather.com/v1/location/EGMC:9:GB/observations/
↪historical.json?apiKey=6532d6454b8aa370768e63d6ba5a832e&units=m"
    date_str = '&startDate=' + date + '&endDate=' + date

    try:
        response = requests.get(api_url + date_str)
    except requests.exceptions.RequestException as e:
         return "Error: {}".format(e)

    weather = []
    for items in response.json()['observations']:
        weather.append((datetime.fromtimestamp(items['valid_time_gmt']),␣
↪items['temp'],
                        items['wspd'], items['wx_phrase']))

    df = pd.DataFrame(weather, columns=['Time', 'Temperature', 'Wind Speed',␣
↪'Conditions'])

    return df
```

[116]:

```python
# retrieve weather data for date window determined by cycle hire data
tmp_df = []
for dates in cycle_df_clean.index.get_level_values('Date').unique():
    tmp_df.append(import_weather_from_api(dates.strftime('%Y%m%d')))

weather_df = pd.concat(tmp_df, ignore_index=True).set_index('Time')
weather_df.head()
```

[116]:

```
                     Temperature  Wind Speed Conditions
Time
2019-06-01 00:50:00         13.0        17.0       Fair
2019-06-01 01:50:00         12.0        15.0       Fair
2019-06-01 02:50:00         12.0        11.0       Fair
2019-06-01 03:20:00         12.0         6.0       Fair
2019-06-01 03:50:00         12.0         6.0       Fair
```

[191]:

```python
print(weather_df.info())
print(weather_df.describe())
```

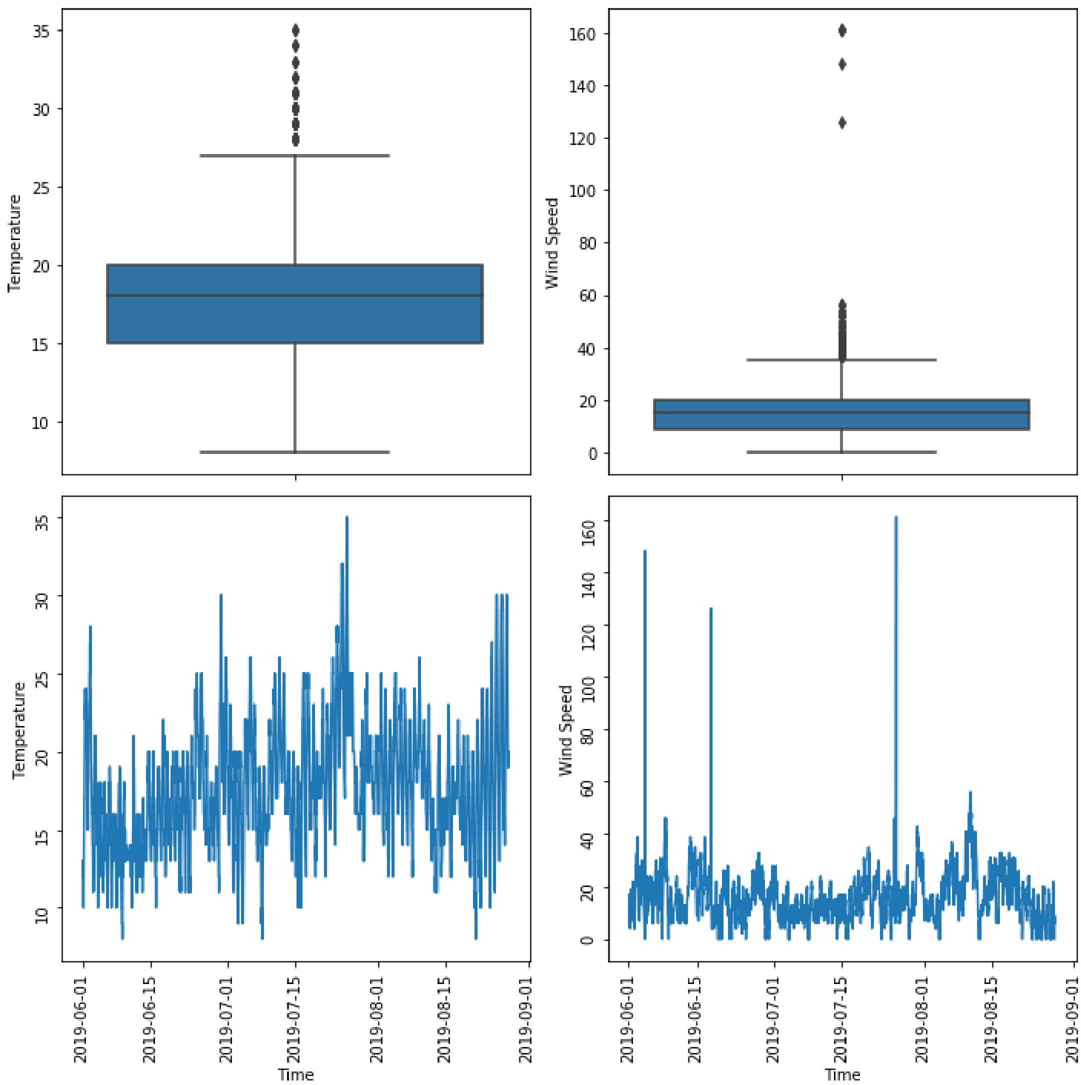
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 3790 entries, 2019-06-01 00:50:00 to 2019-08-27 23:50:00
Data columns (total 3 columns):
Temperature    3789 non-null float64
Wind Speed     3788 non-null float64
Conditions     3790 non-null object
dtypes: float64(2), object(1)
memory usage: 278.4+ KB
```

```
None
       Temperature   Wind Speed
count  3789.000000  3788.000000
mean     17.985220    16.126452
std       4.123911    10.010985
min       8.000000     0.000000
25%      15.000000     9.000000
50%      18.000000    15.000000
75%      20.000000    20.000000
max      35.000000   161.000000
```

We can see from the weather dataframe information and description that there are null values and that the wind speed data could have some outliers. Let's now clean this data!

[221]:

```
fig, ax = plt.subplots(2, 2, figsize=(10,10))
ax1 = sns.boxplot(y=weather_df['Temperature'], ax=ax[0][0])
ax2 = sns.boxplot(y=weather_df['Wind Speed'], ax=ax[0][1])
ax3 = sns.lineplot(x=weather_df.index, y=weather_df['Temperature'], ax=ax[1][0])
ax3.tick_params(labelrotation=90)
ax4 = sns.lineplot(x=weather_df.index, y=weather_df['Wind Speed'], ax=ax[1][1])
ax4.tick_params(labelrotation=90)
fig.tight_layout()
plt.show()
```



[225]:
```
# foward fill na values
weather_df = weather_df.fillna(method='ffill')
if not weather_df.isnull().any().sum():
    print('No more null values')
```

No more null values

[253]:
```
# find outliers in wind speed data by seeing if change in data is large or if␣
↪value is large
diff = weather_df['Wind Speed'].diff()
print('\033[1m' + 'Wind speed data showing large changes in speed:' + '\033[0m')
print(weather_df[diff > 20]['Wind Speed'])
print('\033[1m' + 'Difference in speed was:' + '\033[0m')
```

```
print(diff[diff > 20])
print('\033[1m' + 'Wind speed data with values > 50km/h:' + '\033[0m')
print(weather_df[weather_df['Wind Speed'] > 50])
```

```
Wind speed data showing large changes in speed:
Time
2019-06-04 08:50:00    148.0
2019-06-18 00:20:00    126.0
2019-07-26 04:20:00    161.0
Name: Wind Speed, dtype: float64
Difference in speed was:
Time
2019-06-04 08:50:00    139.0
2019-06-18 00:20:00    111.0
2019-07-26 04:20:00    154.0
Name: Wind Speed, dtype: float64
Wind speed data with values > 50km/h:
                     Temperature  Wind Speed               Conditions
Time
2019-06-04 08:50:00         16.0       148.0             Fair / Windy
2019-06-18 00:20:00         13.0       126.0             Fair / Windy
2019-07-26 04:20:00         22.0       161.0             Fair / Windy
2019-07-26 04:50:00         22.0       161.0             Fair / Windy
2019-07-26 05:20:00         22.0       161.0             Fair / Windy
2019-08-10 12:20:00         20.0        52.0  Showers in the Vicinity
2019-08-10 13:20:00         21.0        52.0    Mostly Cloudy / Windy
2019-08-10 13:50:00         21.0        56.0    Mostly Cloudy / Windy
2019-08-10 14:20:00         21.0        54.0    Mostly Cloudy / Windy
2019-08-10 15:20:00         22.0        56.0    Partly Cloudy / Windy
2019-08-10 15:50:00         22.0        54.0    Partly Cloudy / Windy
2019-08-10 16:20:00         21.0        52.0    Partly Cloudy / Windy
2019-08-10 16:50:00         20.0        52.0    Partly Cloudy / Windy
```

It is clear that we have 5 data errors in the wind speed data series for the values above 100 km/h. The change in wind speed is too large and wind of that magnitude would have definitely made the news which it didn't. A likely explanation could be that the actual wind speed was 1/10th that of the recorded one and there was a logging error with missing the decimal point. Let's look closer at the data aorund the outliers to determine what to do with it.

[259]:
```
print(weather_df.loc['2019-06-04 07:00':'2019-06-04 10:00'])
print(weather_df.loc['2019-06-17 23:00':'2019-06-18 02:00'])
print(weather_df.loc['2019-07-26 03:00':'2019-07-26 06:00'])
```

```
                     Temperature  Wind Speed    Conditions
Time
2019-06-04 07:20:00         14.0         7.0          Fair
2019-06-04 07:50:00         15.0         9.0          Fair
2019-06-04 08:20:00         17.0         9.0          Fair
```

```
2019-06-04 08:50:00         16.0       148.0  Fair / Windy
2019-06-04 09:20:00         18.0        17.0          Fair
2019-06-04 09:50:00         16.0        19.0          Fair
                     Temperature  Wind Speed    Conditions
Time
2019-06-17 23:20:00         14.0        13.0          Fair
2019-06-17 23:50:00         14.0        15.0          Fair
2019-06-18 00:20:00         13.0       126.0  Fair / Windy
2019-06-18 01:20:00         12.0         9.0          Fair
                     Temperature  Wind Speed    Conditions
Time
2019-07-26 03:20:00         22.0         7.0          Fair
2019-07-26 04:20:00         22.0       161.0  Fair / Windy
2019-07-26 04:50:00         22.0       161.0  Fair / Windy
2019-07-26 05:20:00         22.0       161.0  Fair / Windy
2019-07-26 05:50:00         21.0         9.0          Fair
```

Having inspected the data, it seems that the best way to deal with these outliers is to divide it by 10.

[261]:
```
outliers_idx = weather_df[weather_df['Wind Speed'] > 100].index
for i in outliers_idx:
    weather_df.loc[i, 'Wind Speed'] =  weather_df.loc[i, 'Wind Speed']/10
```

Now that we have cleaned the data, we can add weather as a feature to our data set. However, before we do this, note that the conditions feature has 27 different categories. Features that are defined categorically with many different categories add significant complexity to the model. We should try and reduce this to a manageable set without losing the information and accuracy of the data.

[264]:
```
print('The weather conditions are:')
for i, x in enumerate(weather_df['Conditions'].unique()):
    print(i, x)
```

```
The weather conditions are:
0 Fair
1 Fair / Windy
2 Partly Cloudy
3 Rain Shower
4 Light Rain Shower
5 Showers in the Vicinity
6 Light Rain
7 Mostly Cloudy
8 Mostly Cloudy / Windy
9 Light Rain / Windy
10 Partly Cloudy / Windy
11 Light Rain Shower / Windy
12 Rain
13 Thunder in the Vicinity
```

```
14 Light Rain with Thunder
15 T-Storm
16 Mist
17 Heavy T-Storm
18 Cloudy
19 Shallow Fog
20 Light Drizzle
21 Thunder
22 Heavy Rain Shower / Windy
23 T-Storm / Windy
24 Patches of Fog
25 Fog
26 Haze
```

We can reduce it to 4 different categories of weather conditions as such:

1. Good weather: 0, 1, 2, 10, 16, 18, 19
2. OK weather: 4, 5, 7, 8, 11, 13, 20, 24
3. Bad weather: 3, 6, 9, 12, 14, 21, 25
4. Very bad weather: 15, 17, 22, 23, 26

[265]:

```python
# create dictionary map for weather conditions
map_weather = {}
for i, x in enumerate(weather_df['Conditions'].unique()):
    if i in [0, 1, 2, 10, 16, 18, 19]:
        map_weather[x] = 'Good weather'
    elif i in [4, 5, 7, 8, 11, 13, 20, 24]:
        map_weather[x] = 'OK weather'
    elif i in [3, 6, 9, 12, 14, 21, 25]:
        map_weather[x] = 'Bad weather'
    elif i in [15, 17, 22, 23, 26]:
        map_weather[x] = 'Very bad weather'
    else:
        map_weather[x] = np.nan

weather_df['w_cond'] = weather_df['Conditions'].map(map_weather)
weather_df = weather_df.drop(columns='Conditions')
weather_df.head()
```

[265]:

```
                     Temperature  Wind Speed        w_cond
Time
2019-06-01 00:50:00         13.0        17.0  Good weather
2019-06-01 01:50:00         12.0        15.0  Good weather
2019-06-01 02:50:00         12.0        11.0  Good weather
2019-06-01 03:20:00         12.0         6.0  Good weather
2019-06-01 03:50:00         12.0         6.0  Good weather
```

We can further improve peformance of our model by using one hot encoding on the categorical variable, weather condition. Here, we use the pandas dataframe method `get_dummies()` to encode the weather condition data.

[266]:
```
weather_df = pd.get_dummies(weather_df)
weather_df.head()
```

[266]:
```
                     Temperature  Wind Speed  w_cond_Bad weather  \
Time
2019-06-01 00:50:00         13.0        17.0                   0
2019-06-01 01:50:00         12.0        15.0                   0
2019-06-01 02:50:00         12.0        11.0                   0
2019-06-01 03:20:00         12.0         6.0                   0
2019-06-01 03:50:00         12.0         6.0                   0

                     w_cond_Good weather  w_cond_OK weather  \
Time
2019-06-01 00:50:00                    1                  0
2019-06-01 01:50:00                    1                  0
2019-06-01 02:50:00                    1                  0
2019-06-01 03:20:00                    1                  0
2019-06-01 03:50:00                    1                  0

                     w_cond_Very bad weather
Time
2019-06-01 00:50:00                        0
2019-06-01 01:50:00                        0
2019-06-01 02:50:00                        0
2019-06-01 03:20:00                        0
2019-06-01 03:50:00                        0
```

Now, we can merge the weather data with the cycle trips data. Note that the time index is not the same on both dataframes so we cannot merge them directly. We use the pandas index method `get_loc(key, method=nearest)` to find the weather data in our weather dataframe at the time nearest to the trip start time.

[267]:
```
# find weather data by matching to nearest time
weather_dict = {}
for date in cycle_df_clean.index.get_level_values('Start Date').unique():
    weather_dict[date] = weather_df.iloc[weather_df.index.get_loc(date,␣
 ↪method='nearest')]

# merge new weather dataframe with same index as trip data
df = pd.DataFrame.from_dict(weather_dict, orient='index')
cycle_df_weather = cycle_df_clean.merge(df, left_on='Start Date',␣
 ↪right_index=True, how='left')
```

[268]:
```
cycle_df_weather.head()
```

[268]:

```
                                        StartStation Name            End Date  \
Date       Start Date
2019-06-01 2019-06-01  Westminster University, Marylebone 2019-06-01 00:07:00
           2019-06-01          Upcerne Road, West Chelsea 2019-06-01 00:01:00
           2019-06-01          Mile End Stadium, Mile End 2019-06-01 00:19:00
           2019-06-01      Bethnal Green Road, Shoreditch 2019-06-01 00:10:00
           2019-06-01          Mile End Stadium, Mile End 2019-06-01 00:19:00

                                               EndStation Name  \
Date       Start Date
2019-06-01 2019-06-01  St. John's Wood Church, The Regent's Park
           2019-06-01                 Upcerne Road, West Chelsea
           2019-06-01     Mile End Park Leisure Centre, Mile End
           2019-06-01                 Curlew Street, Shad Thames
           2019-06-01     Mile End Park Leisure Centre, Mile End

                       Duration(mins)  Bike Id  StartStation Id  \
Date       Start Date
2019-06-01 2019-06-01             7.0    13485              257
           2019-06-01             1.0    14376              745
           2019-06-01            19.0    10693              712
           2019-06-01            10.0     7390              132
           2019-06-01            19.0    11332              712

                       EndStation Id  Day  hour day_code  Temperature  \
Date       Start Date
2019-06-01 2019-06-01            247    5     0        W         13.0
           2019-06-01            745    5     0        W         13.0
           2019-06-01            763    5     0        W         13.0
           2019-06-01            298    5     0        W         13.0
           2019-06-01            763    5     0        W         13.0

                       Wind Speed  w_cond_Bad weather  w_cond_Good weather  \
Date       Start Date
2019-06-01 2019-06-01        17.0                 0.0                  1.0
           2019-06-01        17.0                 0.0                  1.0
           2019-06-01        17.0                 0.0                  1.0
           2019-06-01        17.0                 0.0                  1.0
           2019-06-01        17.0                 0.0                  1.0

                       w_cond_OK weather  w_cond_Very bad weather
Date       Start Date
2019-06-01 2019-06-01                0.0                      0.0
           2019-06-01                0.0                      0.0
           2019-06-01                0.0                      0.0
           2019-06-01                0.0                      0.0
           2019-06-01                0.0                      0.0
```

[269]:
```
cycle_df_weather.info()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 2980310 entries, (2019-06-01 00:00:00, 2019-06-01 00:00:00) to
(2019-08-27 00:00:00, 2019-08-27 23:57:00)
Data columns (total 16 columns):
StartStation Name           object
End Date                    datetime64[ns]
EndStation Name             object
Duration(mins)              float64
Bike Id                     int64
StartStation Id             int64
EndStation Id               int64
Day                         int64
hour                        int64
day_code                    object
Temperature                 float64
Wind Speed                  float64
w_cond_Bad weather          float64
w_cond_Good weather         float64
w_cond_OK weather           float64
w_cond_Very bad weather     float64
dtypes: datetime64[ns](1), float64(7), int64(5), object(3)
memory usage: 378.9+ MB
```

We now have a clean data set with added features such as day type, hour of day, temperature, wind speed and weather condition. In the next section of the project, we will look for relationships within our data set using data visualisation tools.

3.2 Part 2: Data Storytelling

Part_2_Data_Storytelling

November 29, 2019

0.1 Capstone Project 1: Modelling Cycle Hire Network

0.1.1 Part 2: Data Storytelling

In this section of the project, we will use visualization tools such as bokeh to look at relationships between data. Ideally, we want to see linear realtionships between the number of rides/duration of rides and the features we have selected such as type of day, weather and hour of day.

[33]:
```
import numpy as np
import pandas as pd
from scipy import stats

from bokeh.io import output_notebook, show
from bokeh.plotting import figure, output_file, save
from bokeh.layouts import gridplot
from bokeh.models import ColumnDataSource
from matplotlib import pyplot as plt
output_notebook()
```

We will first look at the relationship between number of hires per hour and weather data such as temperature, wind speed and type of weather. Before we get to plotting, we will first define a plotting function, `linear_reg_gridplot()` that will plot the scatter plot and histogram of any two varaiables. This function will also use `np.polyfit()` to fit the data using linear regression and display the mean-squared error(MSE) of the fit on the plot.

[2]:
```
#download cleaned cycle hire data
cycle_df = pd.read_csv('cycle_df_weather.csv', parse_dates=[0, 1, 3])
```

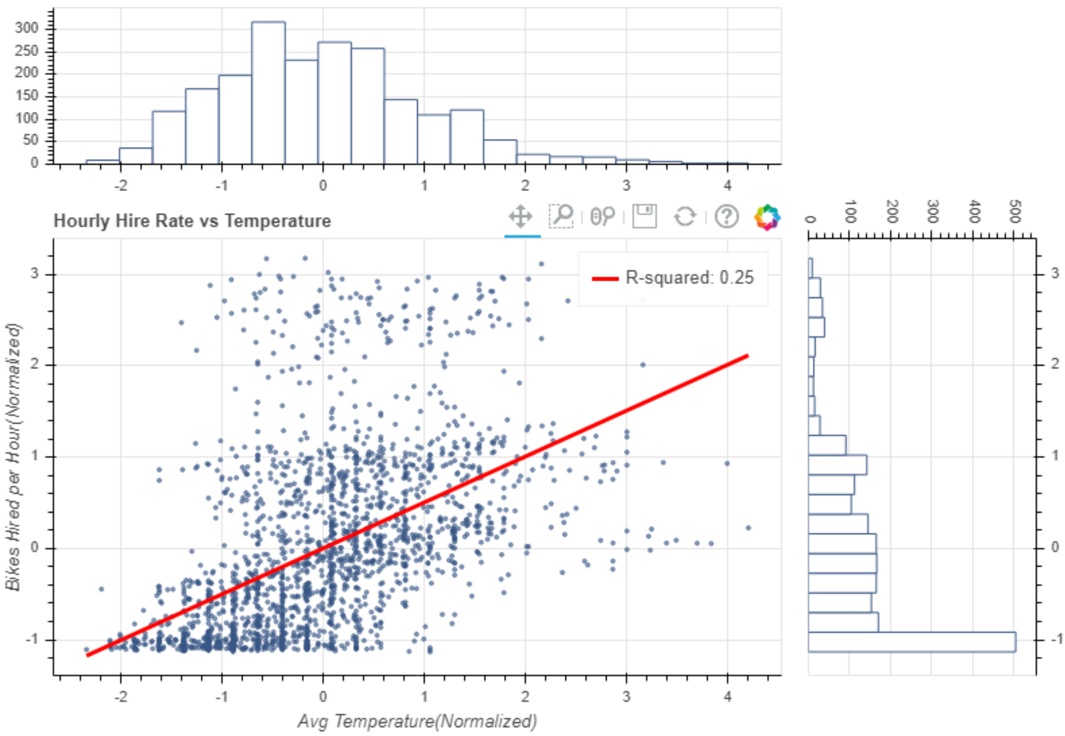
[45]:
```
def r_squared(SSres, y):

    SStot = np.sum(np.square(y - np.mean(y)))
    return round(1 - SSres/SStot, 2)

def linear_reg_gridplot(x, y, title, xlabel, ylabel, normalize_values=True):
    """plots scatter, regression line and histograms of 2 variables"""

    #normalize x, y
    if normalize_values == True:
```

```
        y = (y - np.mean(y))/np.std(y)
        x = (x - np.mean(x))/np.std(x)

    # compute the linear regression line for dataset
    fit = np.polyfit(x, y, 1, full=True)
    x_reg = [min(x), max(x)]
    y_reg = [fit[0][0]*i + fit[0][1] for i in x_reg]

    # plot scatter plot and fitted line
    p = figure(toolbar_location='above', plot_width=600, plot_height=400,␣
↪title=title)
    p.scatter(x, y, size=3, color="#3A5785", alpha=0.6)
    p.line(x_reg, y_reg, color='red', line_width=3, legend='R-squared: ' +␣
↪str(r_squared(fit[1][0], y)))
    p.xaxis.axis_label = xlabel
    p.yaxis.axis_label = ylabel

    # plot horizontal histogram - distribution of x values
    hhist, hedges = np.histogram(x, bins=20)
    hzeros = np.zeros(len(hedges)-1)
    ph = figure(toolbar_location=None, plot_width=p.plot_width, plot_height=150,␣
↪x_range=p.x_range,
            y_range=(0, max(hhist)*1.1), min_border=10, min_border_left=50,␣
↪y_axis_location="left")
    ph.quad(bottom=0, left=hedges[:-1], right=hedges[1:], top=hhist,␣
↪color='white', line_color="#3A5785")

    # plot vertical histogram - distribution of y values
    vhist, vedges = np.histogram(y, bins=20)
    vzeros = np.zeros(len(vedges)-1)
    pv = figure(toolbar_location=None, plot_width=200, plot_height=p.
↪plot_height, x_range=(0, max(vhist)*1.1),
            y_range=p.y_range, min_border=10, x_axis_location="above",␣
↪y_axis_location="right")
    pv.xaxis.major_label_orientation = -np.pi/2
    pv.quad(left=0, bottom=vedges[:-1], top=vedges[1:], right=vhist,␣
↪color="white", line_color="#3A5785")


    layout = gridplot([[ph, None], [p, pv]], merge_tools=False)
    return(layout)
```

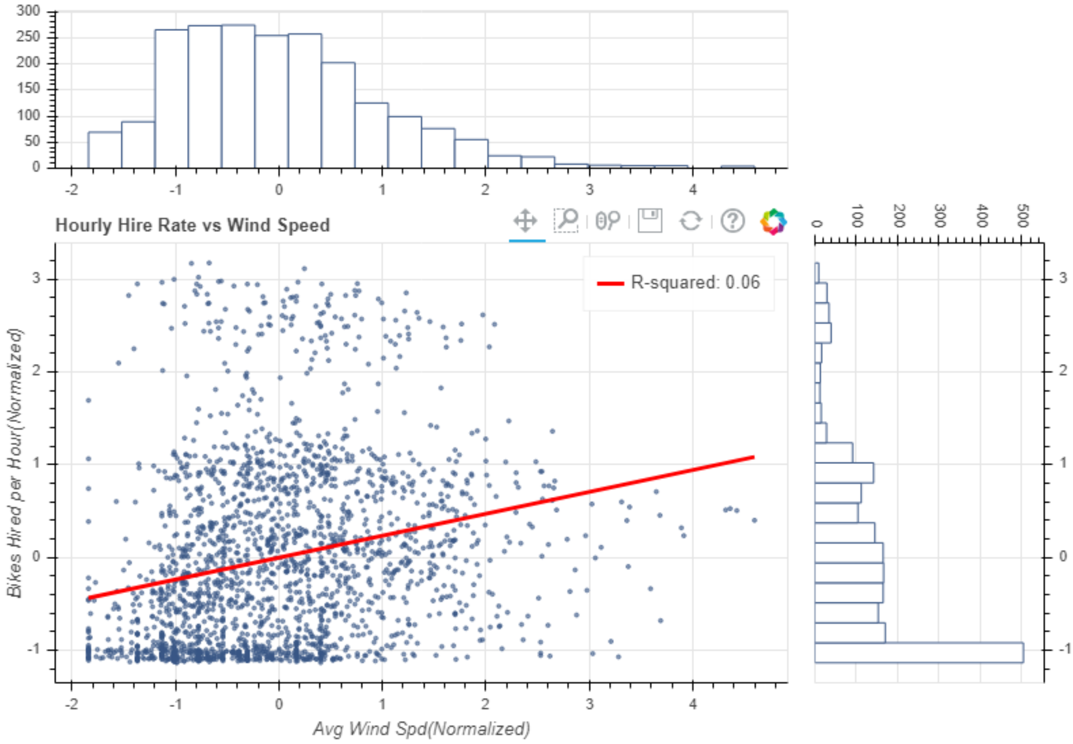
[5]:
```
hourly_hire_group = cycle_df.groupby(['Date', 'hour'])
```

See plots below showing relationship between weather data and hourly ride count. We can see that the hourly hire rate is positively correlated with temperature and good weather conditions. However, there is not enough evidence to show correlation between wind speed and ride count.

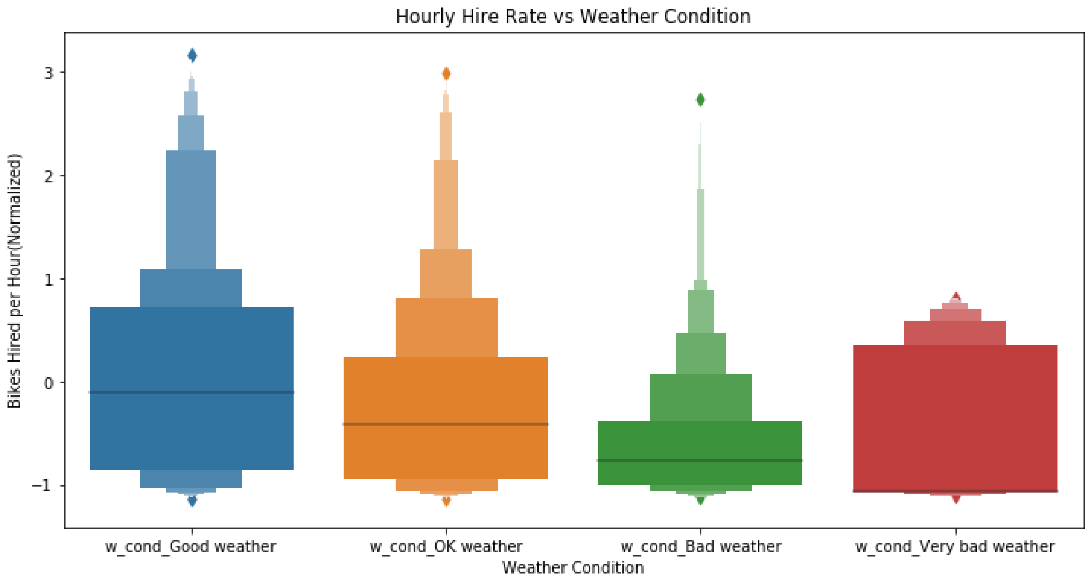
[46]:
```
# plot relationship between hourly cycle hire count and average temperature
plot = linear_reg_gridplot(hourly_hire_group.mean()['Temperature'],␣
↪hourly_hire_group.count()['Duration(mins)'],
                    'Hourly Hire Rate vs Temperature', 'Avg␣
↪Temperature(Normalized)', 'Bikes Hired per Hour(Normalized)')
show(plot)
output_file("images/temp_relationship.html")
save(plot)
```



[47]:
```
# plot relationship between hourly cycle hire count and average wind speed
plot = linear_reg_gridplot(hourly_hire_group.mean()['Wind Speed'],␣
↪hourly_hire_group.count()['Duration(mins)'],
                    'Hourly Hire Rate vs Wind Speed', 'Avg Wind Spd(Normalized)',␣
↪'Bikes Hired per Hour(Normalized)')
show(plot)
output_file("images/wind_relationship.html")
save(plot)
```



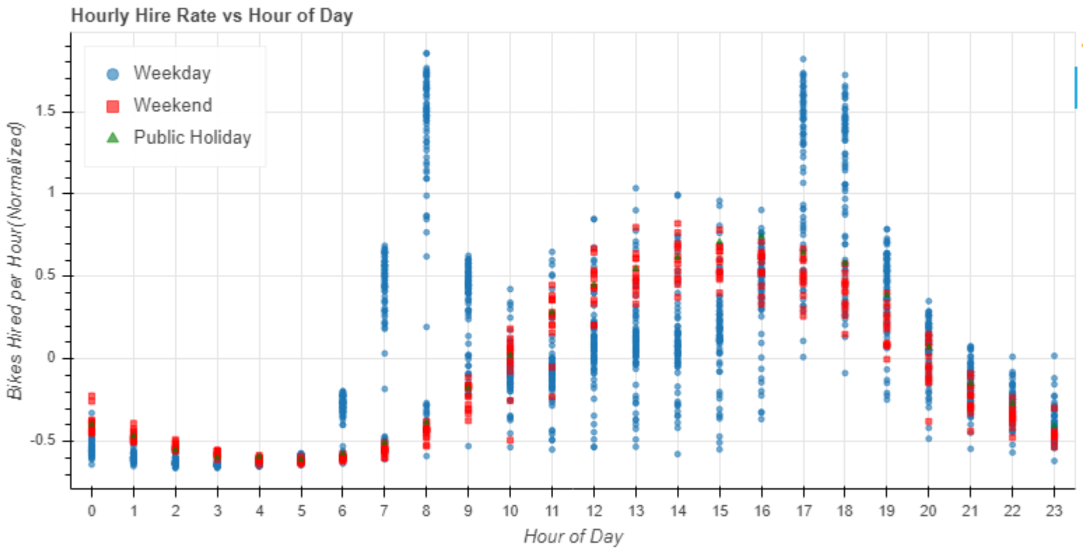
[21]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# set up dataframe with just 2 columns: weather type and hourly ride rate -␣
↪easier to plot
weather_cond = hourly_hire_group.mean()[['w_cond_Bad weather', 'w_cond_Good␣
↪weather', 'w_cond_OK weather', 'w_cond_Very bad weather']].idxmax(axis=1)
hourly_count = hourly_hire_group.count()['Duration(mins)']
hourly_count_norm = (hourly_count - np.mean(hourly_count))/np.std(hourly_count)
df = pd.concat([weather_cond, hourly_count_norm], axis=1).rename(columns={0:␣
↪'weather'})

plt.figure(figsize=(12,6))
plt.title('Hourly Hire Rate vs Weather Condition')
sns_plot = sns.boxenplot(x="weather", y="Duration(mins)", data=df)
plt.xlabel('Weather Condition')
plt.ylabel('Bikes Hired per Hour(Normalized)')
plt.show()
sns_plot.figure.savefig("images/wcond_relationship.png")
```



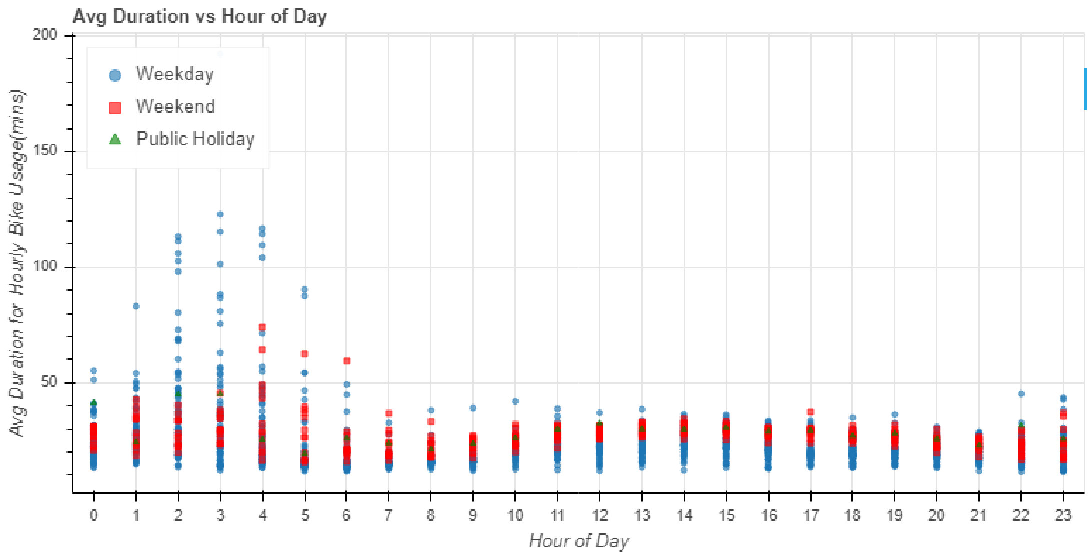
We can now look at how the type of day affects the number of rides and duration of rides. Here, we use an interactive legend for the type of day and look at the data by hour. We can see that number of bikes hired is greater on a weekday with peak hiring times around morning and evening peak hours. As for the average duration, the peaks are around late night or early morning when alternative transport modes are limited.

[39]:
```
def plot_scatter_by_day(data, ylabel, title):

    source1 = ColumnDataSource(data[data.day_code == 'W'])
    source2 = ColumnDataSource(data[data.day_code == 'WD'])
    source3 = ColumnDataSource(data[data.day_code == 'PH'])

    p = figure(x_range=data.hour.unique(), plot_width=800, plot_height=400,␣
↪title=title)
    p.circle(x='hour', y='Duration(mins)', source=source1, alpha=0.6,␣
↪legend='Weekday')
    p.square(x='hour', y='Duration(mins)', source=source2, alpha=0.6,␣
↪legend='Weekend', color='red')
    p.triangle(x='hour', y='Duration(mins)', source=source3, alpha=0.6,␣
↪legend='Public Holiday', color='green')

    p.legend.location = "top_left"
    p.legend.click_policy="hide"
    p.xaxis.axis_label = 'Hour of Day'
    p.yaxis.axis_label = ylabel

    return(p)
```

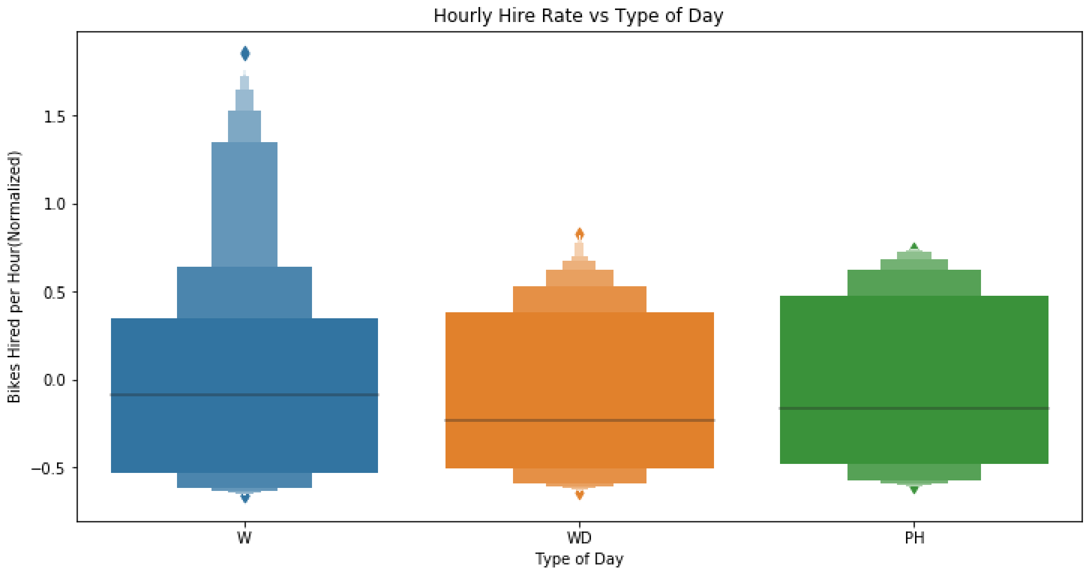
[40]:
```
day_group = cycle_df.groupby(['Date', 'hour', 'day_code'])
data = day_group.count()['Duration(mins)'].reset_index()
data.hour = data.hour.astype(str)
duration = data['Duration(mins)']
data['Duration(mins)'] = (duration -np.mean(duration))/len(duration)

plot = plot_scatter_by_day(data, 'Bikes Hired per Hour(Normalized)', 'Hourly␣
↪Hire Rate vs Hour of Day')
show(plot)
output_file("images/hour_relationship.html")
save(plot)
```



[41]:
```
data_time_avg = day_group.mean()['Duration(mins)'].reset_index()
data_time_avg.hour = data_time_avg.hour.astype(str)

plot = plot_scatter_by_day(data_time_avg, 'Avg Duration for Hourly Bike␣
↪Usage(mins)', 'Avg Duration vs Hour of Day')
show(plot)
output_file("images/hour_dur_relationship.html")
save(plot)
```

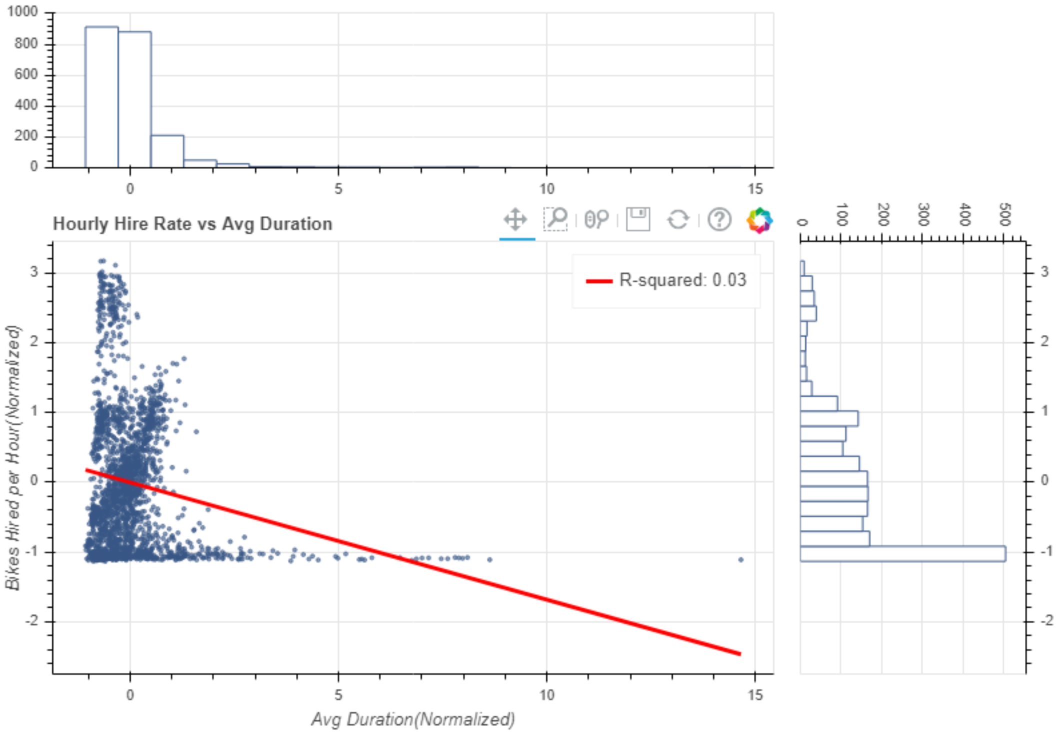


[20]:
```
plt.figure(figsize=(12,6))
plt.title('Hourly Hire Rate vs Type of Day')
sns_plot = sns.boxenplot(x="day_code", y="Duration(mins)", data=data)
plt.xlabel('Type of Day')
plt.ylabel('Bikes Hired per Hour(Normalized)')
plt.show()
sns_plot.figure.savefig("images/day_type_relationship.png")
```



In the previous plots, we suspect that the average duration and the number of rides might be inversely proportional. Below, we try to visualise the relationship between the 2 variables. We can see that the higher hourly bike hire rates are concentrated towards lower average durations.

[42]:
```
plot = linear_reg_gridplot(hourly_hire_group.mean()['Duration(mins)'],␣
↪hourly_hire_group.count()['Duration(mins)'],
                    'Hourly Hire Rate vs Avg Duration', 'Avg␣
↪Duration(Normalized)', 'Bikes Hired per Hour(Normalized)')
show(plot)
output_file("images/count_dur_relationship.html")
save(plot)
```



3.3 Part 3 & 4: Feature Engineering & Statistical Analysis

Part_3_Feature_Engineering_&_Stats

November 29, 2019

0.1 Capstone Project 1: Modelling Cycle Hire Network

0.1.1 Part 3: Feature Engineering

In part 3, we will contine our work in Part 1 in cleaning up the data and selecting the right features for our model.

[4]:
```
import pandas as pd
import numpy as np
from datetime import datetime
from datetime import timedelta
```

[5]:
```
#download cleaned cycle hire data
cycle_df = pd.read_csv('cycle_df_weather.csv', parse_dates=[0, 1, 3])
```

We want to use our model to predict the next-day demand at each station. We need to first reduce our observations to be a daily aggregation, then include more predictor variables such as rides count from day before and 7 day moving average for the duration.

[169]:
```
# drop non useful columns
features_df = cycle_df.drop(columns=['Start Date', 'End Date', 'StartStation␣
↪Name', 'EndStation Name',
                                    'EndStation Id', 'hour'])

# aggregate data by day and station
aggfunc = {'Duration(mins)' : 'mean', 'Day' : 'max', 'day_code' : 'max',␣
↪'Temperature' : 'mean', 'Wind Speed' : 'mean',
           'w_cond_Bad weather' : 'mean', 'w_cond_Good weather' : 'mean',␣
↪'w_cond_OK weather' : 'mean',
           'w_cond_Very bad weather' : 'mean', 'Bike Id': 'count'}

features_df = features_df.groupby(['StartStation Id', 'Date']).agg(aggfunc).
↪rename(columns={'Bike Id' : 'Count'})

# transform day codes into binary codes
features_df['day_code'] = features_df['day_code'].replace({'PH' : 0, 'WD' : 0,␣
↪'W': 1})
```

Before computing the rides count for the day before and 7 day rolling average for duration, we need to first fill in values for dates not found in the observations i.e. dates where no one hired a cycle. We can do this by creating a new index that spans all the stations for all the dates in the dataset and use reindexing to fill the days with 0 for number of rides and 0 for duration of rides.

[170]:
```
#creating new index for all dates
date_range = np.tile(pd.date_range(start = features_df.index.min()[1], end=␣
↪features_df.index.max()[1]),
             features_df.index.max()[0])
num_range = np.repeat((np.array(list(range(features_df.index.max()[0]))) + 1),
              (features_df.index.max()[1] - features_df.index.min()[1]).days +␣
↪1)
new_index = list(zip(num_range, date_range))

new_index_df = features_df[['Duration(mins)', 'Count']].reindex(new_index).
↪fillna(0)

#adding new features: day before count and 7 day rolling duration average
new_index_df['day_bf_count'] = new_index_df['Count'].groupby(level=0).shift()
new_index_df['7d_rolling_dur'] = new_index_df['Duration(mins)'].groupby(level=0).
↪apply(lambda x: x.rolling(window=7).mean())
```

[171]:
```
features_df['day_bf_count'] = new_index_df[new_index_df['Count'] !=␣
↪0]['day_bf_count']
features_df['7d_rolling_dur'] = new_index_df[new_index_df['Count'] !=␣
↪0]['7d_rolling_dur']
```

Another useful information might be the number of rides that end at the specific station. This way we can see if there is a demand-supply mismatch.

[172]:
```
end_station_num = cycle_df.groupby(['EndStation Id', 'Date']).count()['Bike Id']
end_station_num = end_station_num.rename_axis(index={"EndStation Id":
↪"StartStation Id"})
end_station_num = end_station_num.reindex(new_index).fillna(0).to_frame()

end_station_num['day_bf_count_end'] = end_station_num.groupby(level=0).shift()
features_df = features_df.merge(end_station_num, how='left', left_index=True,␣
↪right_on=['StartStation Id', 'Date']).rename(columns={'Bike Id':'count_end'})
```

[173]:
```
new_col_names = ['Day', 'day_code', 'day_bf_count', 'day_bf_count_end',␣
↪'7d_rolling_dur', 'Temperature', 'Wind Speed',
                'w_cond_Good weather', 'w_cond_OK weather', 'w_cond_Bad␣
↪weather', 'w_cond_Very bad weather']
Y = features_df[['Duration(mins)', 'Count', 'count_end']]
features_df = features_df.drop(columns=['Duration(mins)', 'Count', 'count_end']).
↪reindex(columns=new_col_names)

features_df.head()
```

[173]:

```
                            Day  day_code  day_bf_count  day_bf_count_end  \
StartStation Id Date
1               2019-06-01    5         1           NaN               NaN
                2019-06-02    6         0          16.0              14.0
                2019-06-03    0         1          34.0              22.0
                2019-06-04    1         1          23.0              13.0
                2019-06-05    2         1          31.0              13.0

                            7d_rolling_dur  Temperature  Wind Speed  \
StartStation Id Date
1               2019-06-01             NaN    21.812500   15.562500
                2019-06-02             NaN    24.352941   27.588235
                2019-06-03             NaN    17.043478   23.391304
                2019-06-04             NaN    15.677419   12.000000
                2019-06-05             NaN    15.300000   19.650000

                            w_cond_Good weather  w_cond_OK weather  \
StartStation Id Date
1               2019-06-01             1.000000           0.000000
                2019-06-02             1.000000           0.000000
                2019-06-03             1.000000           0.000000
                2019-06-04             0.741935           0.258065
                2019-06-05             1.000000           0.000000

                            w_cond_Bad weather  w_cond_Very bad weather
StartStation Id Date
1               2019-06-01                 0.0                      0.0
                2019-06-02                 0.0                      0.0
                2019-06-03                 0.0                      0.0
                2019-06-04                 0.0                      0.0
                2019-06-05                 0.0                      0.0
```

[191]:

```
features_df.rename(columns={'day_code':'is_weekday', 'Temperature':
 ↪'temperature', 'Wind Speed':'wind_speed',
                           'w_cond_Good weather':'good_weather', 'w_cond_OK␣
 ↪weather':'ok_weather',
                           'w_cond_Bad weather':'bad_weather', 'w_cond_Very bad␣
 ↪weather':'very_bad_weather',
                           'Day':'day_no', 'day_bf_count':'start_count_day_bf',␣
 ↪'day_bf_count_end':'end_count_day_bf'},
                  inplace=True)
```

[192]:

```
features_df.head()
```

[192]:

```
                            day_no  is_weekday  start_count_day_bf  \
StartStation Id Date
1               2019-06-01       5           1                 NaN
```

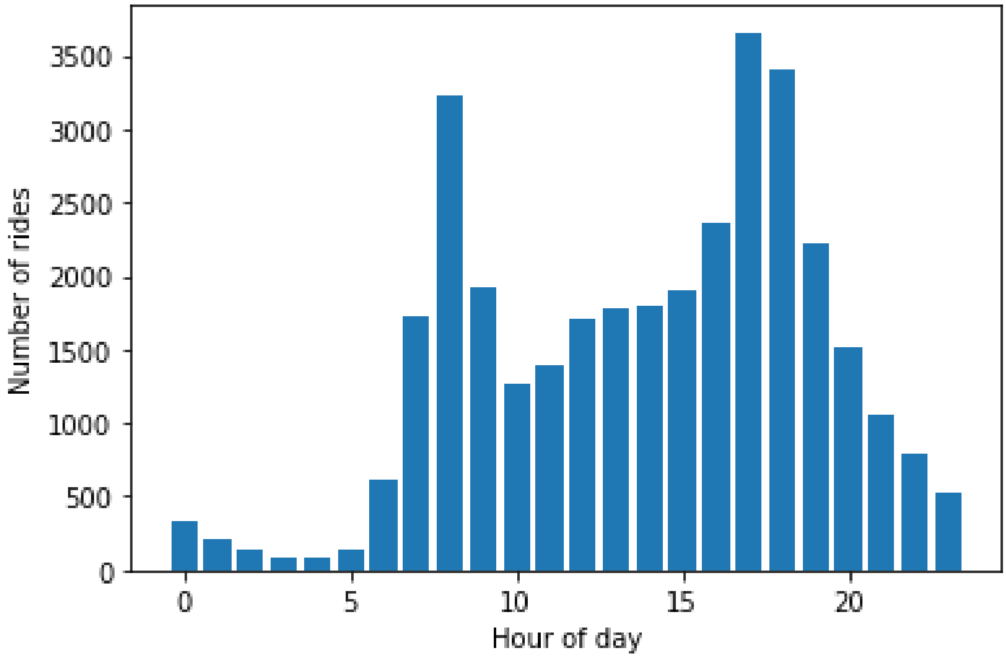
```
                2019-06-02        6           0               16.0
                2019-06-03        0           1               34.0
                2019-06-04        1           1               23.0
                2019-06-05        2           1               31.0

                           end_count_day_bf  7d_rolling_dur  temperature  \
StartStation Id Date
1               2019-06-01              NaN             NaN    21.812500
                2019-06-02             14.0             NaN    24.352941
                2019-06-03             22.0             NaN    17.043478
                2019-06-04             13.0             NaN    15.677419
                2019-06-05             13.0             NaN    15.300000

                           wind_speed  good_weather  ok_weather  bad_weather  \
StartStation Id Date
1               2019-06-01  15.562500      1.000000    0.000000          0.0
                2019-06-02  27.588235      1.000000    0.000000          0.0
                2019-06-03  23.391304      1.000000    0.000000          0.0
                2019-06-04  12.000000      0.741935    0.258065          0.0
                2019-06-05  19.650000      1.000000    0.000000          0.0

                           very_bad_weather
StartStation Id Date
1               2019-06-01              0.0
                2019-06-02              0.0
                2019-06-03              0.0
                2019-06-04              0.0
                2019-06-05              0.0
```

0.1.2 Part 4: Statistical Analysis

[1]:
```
import pymc3 as pm
import theano.tensor as tt
from matplotlib import pyplot as plt
import matplotlib.dates as mdates
```

Q1. How does the frequency of hiring bikes change in a day?

[6]:
```
count_data = cycle_df.groupby(['Date', 'hour']).count()['Bike Id'].
 ↪groupby('hour').mean()
plt.bar(count_data.index, count_data.values)
plt.xlabel('Hour of day')
plt.ylabel('Number of rides')
plt.show()
```



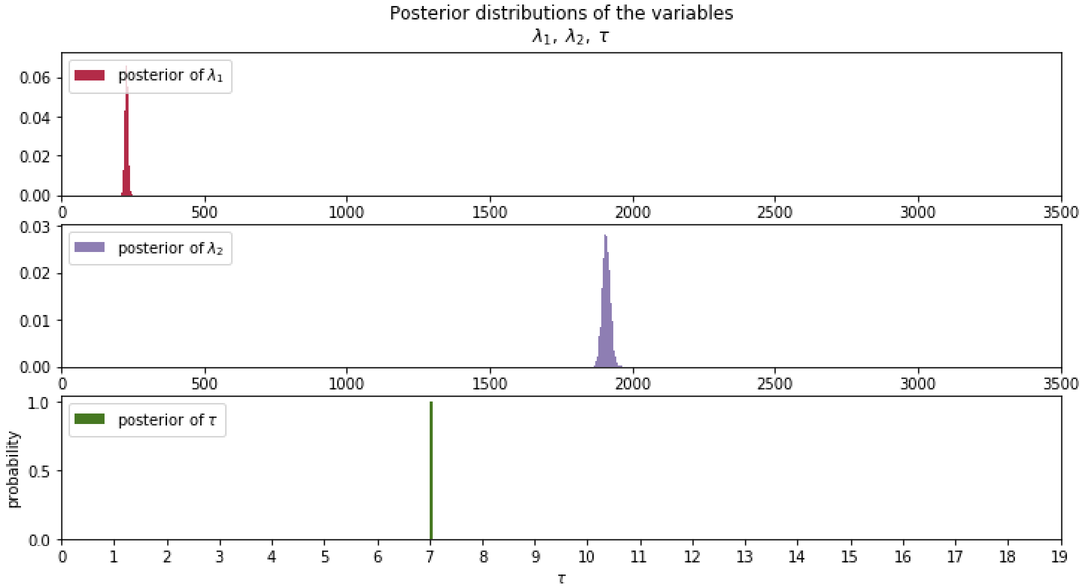
[32]:
```
count_data = count_data.loc[0:19]

with pm.Model() as model:
    alpha = 1.0/count_data.values.mean()
    lambda_1 = pm.Exponential("lambda_1", alpha)
    lambda_2 = pm.Exponential("lambda_2", alpha)
    lambda_3 = pm.Exponential("lambda_3", alpha)

    tau_1 = pm.DiscreteUniform("tau_1", lower=0, upper=len(count_data) - 1)
    tau_2 = pm.DiscreteUniform("tau_2", lower=tau_1, upper=len(count_data) - 1)

    idx = np.arange(len(count_data))
    lambda_ = pm.math.switch(tau_2 > idx, pm.math.switch(tau_1 > idx, lambda_1,␣
 ↪lambda_2), lambda_3)

    observation = pm.Poisson("obs", lambda_, observed=count_data.values)
```

[33]:
```
with model:
    step = pm.Metropolis()
    trace = pm.sample(10000, tune=5000,step=step)

lambda_1_samples = trace['lambda_1']
lambda_2_samples = trace['lambda_2']
lambda_3_samples = trace['lambda_3']
```

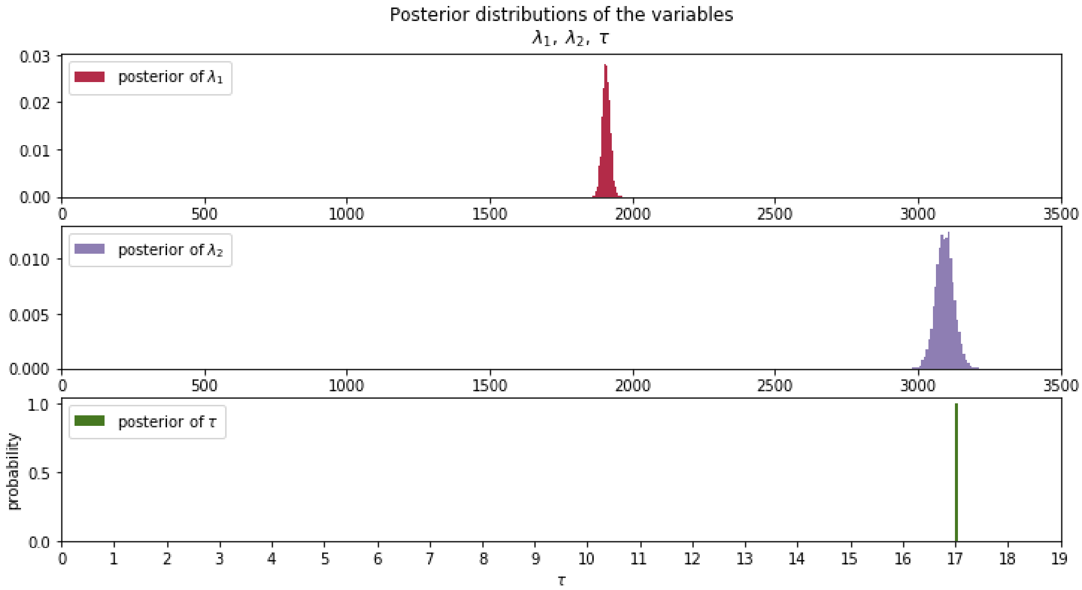
```
tau_samples = trace['tau_1']
tau_2_samples = trace['tau_2']
```

```
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>Metropolis: [tau_2]
>Metropolis: [tau_1]
>Metropolis: [lambda_3]
>Metropolis: [lambda_2]
>Metropolis: [lambda_1]
Sampling 4 chains: 100% 60000/60000 [00:36<00:00, 1648.91draws/s]
The number of effective samples is smaller than 25% for some parameters.
```

[36]:
```
def plot_tau_dist(lambda_1, lambda_2, tau, n_tau, xlim, fig_size=(12, 6)):

    plt.figure(figsize=fig_size)
    ax = plt.subplot(311)
    plt.hist(lambda_1, bins=30, alpha=0.85,
             label="posterior of $\lambda_1$", color="#A60628", density=True)
    plt.legend(loc="upper left")
    plt.title(r"""Posterior distributions of the variables
    $\lambda_1,\;\lambda_2,\;\tau$""")
    plt.xlabel("$\lambda_1$ value")
    plt.xlim(xlim)

    ax = plt.subplot(312)
    plt.hist(lambda_2, bins=30, alpha=0.85,
             label="posterior of $\lambda_2$", color="#7A68A6", density=True)
    plt.legend(loc="upper left")
    plt.xlabel("$\lambda_2$ value")
    plt.xlim(xlim)

    plt.subplot(313)
    w = 1.0 / tau.shape[0] * np.ones_like(tau)
    plt.hist(tau, bins=n_tau, alpha=1,
             label=r"posterior of $\tau$",
             color="#467821", weights=w, rwidth=2.)
    plt.xticks(np.arange(n_tau))

    plt.legend(loc="upper left")
    plt.xlabel(r"$\tau$")
    plt.ylabel("probability")

plot_tau_dist(lambda_1_samples, lambda_2_samples, tau_samples, len(count_data),␣
 ↪[0, 3500])
```



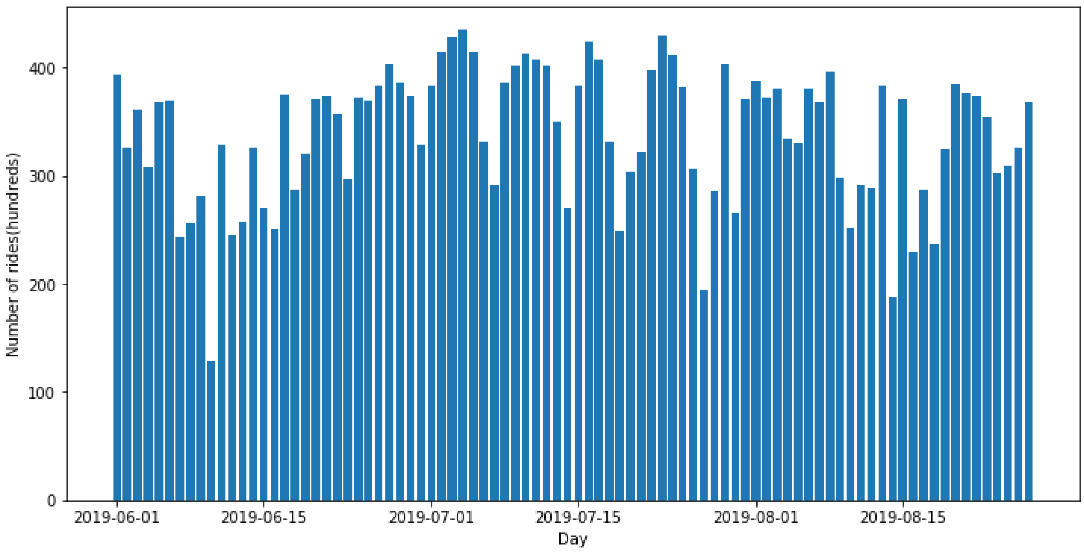
[37]:
```
plot_tau_dist(lambda_2_samples, lambda_3_samples, tau_2_samples, len(count_data), [0, 3500])
```



The model shows that there is a close to 100% probability that the ride count changes at 7am and at 5pm. The posterior distributions of the 2 λs are very distinct indicating there is a significant change in ridership after 7am and after 5pm with the mean of λ_1 at around 250, the mean of λ_2 close to 2000 and the mean of λ_3 close to 3000.

Q2. Did the frequency of bike hiring change during significantly during this time period?

[110]:
```
count_data = cycle_df.groupby(['Date']).count()['Bike Id']/100
plt.figure(figsize=(12,6))
plt.bar(count_data.index, count_data.values)
plt.format_xdata = mdates.DateFormatter('%Y-%m-%d')
plt.xlabel('Day')
plt.ylabel('Number of rides(hundreds)')
plt.show()
```



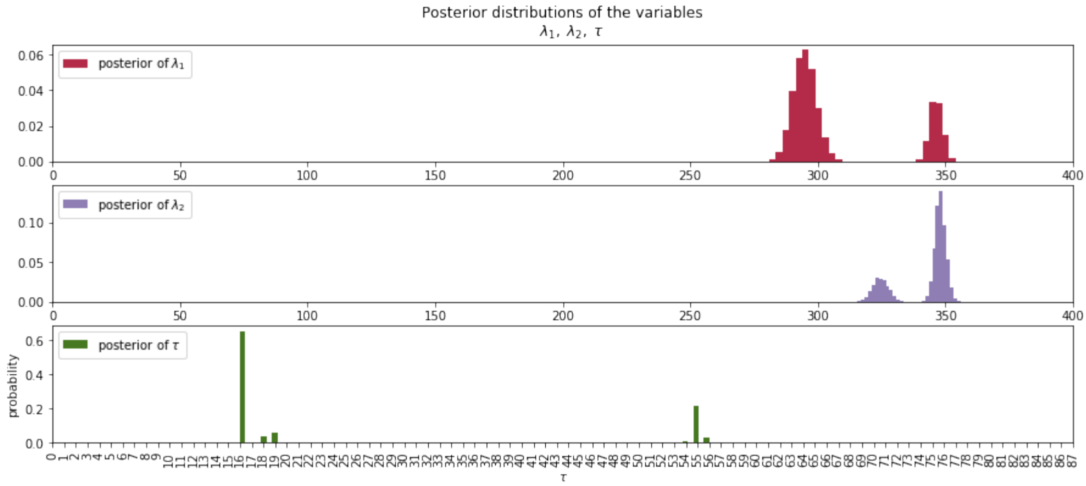
[111]:
```
with pm.Model() as model:
    alpha = 1.0/count_data.values.mean()
    lambda_1 = pm.Exponential("lambda_1", alpha)
    lambda_2 = pm.Exponential("lambda_2", alpha)

    tau = pm.DiscreteUniform("tau", lower=0, upper=len(count_data) - 1)

    idx = np.arange(len(count_data))
    lambda_ = pm.math.switch(tau > idx, lambda_1, lambda_2)

    observation = pm.Poisson("obs", lambda_, observed=count_data.values)

with model:
    step = pm.Metropolis()
    trace = pm.sample(10000, tune=5000,step=step)

lambda_1_samples = trace['lambda_1']
lambda_2_samples = trace['lambda_2']
```

```
tau_samples = trace['tau']
```

```
Multiprocess sampling (4 chains in 4 jobs)
CompoundStep
>Metropolis: [tau]
>Metropolis: [lambda_2]
>Metropolis: [lambda_1]
Sampling 4 chains: 100% 60000/60000 [00:29<00:00, 2011.15draws/s]
The gelman-rubin statistic is larger than 1.4 for some parameters. The sampler
did not converge.
The estimated number of effective samples is smaller than 200 for some
parameters.
```

[112]:

```
plot_tau_dist(lambda_1_samples, lambda_2_samples, tau_samples, len(count_data),␣
↪[0, 400], (15, 6))
plt.xticks(rotation=90)
plt.show()
```



The analysis shows no clear evidence of there being a significant change in ride hiring over the period. There is no distinct differences between λs and the sampler did not converge.

List of Figures

1.1 Location of cycle stations 3
1.2 Number of rides and average duration per day 4
1.3 Sample data of each trip 4

2.1 Relationship between weather conditions and number of rides 6
2.2 Relationship between type of day and number of rides 7
2.3 Relationship between hour of day and ride count and ride duration 7
2.4 Relationship between ride count and duration of rides 8

List of Tables

1.1 Features for model 5